CS135 Tutorial 07

Recursion Patterns

Simple Recursion -> Accumulative Recursion

```
;; (sum-sr n) produces the sum of the numbers from 0 to n, inclusive.
;; sum-sr: Nat -> Nat
(define (sum-sr n)
  (cond [(zero? n) 0]
        [else (+ n (sum-sr (sub1 n)))]))

(check-expect (sum-sr 3) (+ 0 1 2 3))

(sum-sr 10) =>
(+ 10 (sum-sr 9)) =>
(+ 10 (+ 9 (sum-sr 8))) =>
(+ 10 (+ 9 (+ 8 (sum-sr 7)))) =>*
(+ 10 (+ 9 (+ 8 (+ 7 (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 (sum-sr 0)))))))))))) =>
(+ 10 (+ 9 (+ 8 (+ 7 (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0)))))))))) =>
(+ 10 (+ 9 (+ 8 (+ 7 (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 1)))))))))) =>* 55
```

Simple Recursion -> Accumulative Recursion

```
;; (sum-ar n) produces the sum of the numbers from 0 to n, inclusive.
;; sum-ar: Nat -> Nat
(define (sum-ar n) (sum-ar/acc ...))

(check-expect (sum-ar 3) (+ 0 1 2 3))

(define (sum-ar/acc ...) ...)
```

Simple Recursion -> Accumulative Recursion

```
;; (sum-ar n) produces the sum of the numbers from 0 to n, inclusive.
;; sum-ar: Nat -> Nat
(define (sum-ar n) (sum-ar/acc n 0))

(check-expect (sum-ar 3) (+ 0 1 2 3))

;; (sum-ar/acc n sum-so-far) produces the sum from 0 to n + sum-so-far
;; sum-ar/acc: Nat Nat -> Nat
(define (sum-ar/acc n sum-so-far)
  (cond [(zero? n) sum-so-far]
        [else (sum-ar/acc (sub1 n) (+ n sum-so-far))]))

(check-expect (sum-ar/acc 2 3) 6)
```

Simple Recursion -> Accumulative Recursion

```
(sum-ar 10) =>
(sum-ar/acc 10 0) =>
(sum-ar/acc 9 10) =>
(sum-ar/acc 8 19) =>
(sum-ar/acc 7 27) =>
(sum-ar/acc 6 34) =>
(sum-ar/acc 5 40) =>
(sum-ar/acc 4 45) =>
(sum-ar/acc 3 49) =>
(sum-ar/acc 2 52) =>
(sum-ar/acc 1 54) =>
(sum-ar/acc 0 55) =>
55
```

Binary Search Trees

- In class, you covered the concept of a binary search tree.
- In A06 you will write a function, `full?`, a function to determine whether every node had either 0 or 2 children.
- Fullness is one definition of balance for a tree. Balanced trees are usually easier to search and are more efficient
- Today:
  - We will discuss a stricter definition of balanced search trees
  - Learn how to build a balanced binary search tree

Balanced Binary Trees

- There are several definitions of “balanced”. Here’s one:

- A binary tree is balanced if:
  - The number of nodes in the left and the right subtrees differ by at most 1
  - Both subtrees are also balanced.
  - An empty tree is balanced.

Balanced BST Data Definition

```
(define-struct node (key left right)
;; A Node is a (make-node Nat BalBST BalBST)
;; requires: all keys in left < key
;;           all keys in right > key
;;           |(# nodes in left) - (# nodes in right)| <= 1

;; A balanced binary tree (BalBST) is one of:
;; * empty
;; * Node
```

Building balanced binary search trees

Given a sorted list of number, build a balanced binary search tree.

```
(define-struct node (key left right))

;; (build-bal-bst slon) builds a balanced binary search tree from slon.
;; build-bal-bst: (listof Num) -> BalBST
;; requires: slon is sorted in increasing order
(define (build-bal-bst slon) ...)

(check-expect (build-bal-bst empty) empty)
(check-expect (build-bal-bst (list 1)) (make-node 1 empty empty))
(check-expect (build-bal-bst (list 1 2 3 4 5 6))
     (make-node 4
          (make-node 2 (make-node 1 empty empty) (make-node 3 empty empty))
               (make-node 6 (make-node 5 empty empty) empty)))
```

Required helper functions

```
;; (nth-elem lst n) produces the nth element in lst (counting from 0).
;; nth-elem: (listof X) Nat -> X
(define (nth-elem lon n)
  (cond [(zero? n) (first lon)]
        [else (nth-elem (rest lon) (sub1 n))]))

;; (take lon n) produces a list from the first n elements f lst.
;; take: (listof X) Nat -> (listof X)
(define (take lon n)
  (cond [(zero? n) empty]
        [else (cons (first lon) (take (rest lon) (sub1 n)))]))

;; (drop lon n) produces a list from the elements after the first n+1 elements
(define (drop lon n)
  (cond [(zero? n) (rest lon)]
        [else (drop (rest lon) (sub1 n))]))
```

Required helper functions

```
(define lst (list 0 1 2 3))

(check-expect (nth-elem lst 0) 0)
(check-expect (nth-elem lst 1) 1)
(check-expect (nth-elem lst 3) 3)
(check-expect (take lst 0) empty)
(check-expect (take lst 1) (list 0))
(check-expect (drop lst 0) (list 1 2 3))
(check-expect (drop lst 1) (list 2 3))
(check-expect (drop lst 3) empty)

(check-expect (append (take lst 0) (list (nth-elem lst 0)) (drop lst 0)) lst)
(check-expect (append (take lst 1) (list (nth-elem lst 1)) (drop lst 1)) lst)
(check-expect (append (take lst 2) (list (nth-elem lst 2)) (drop lst 2)) lst)
(check-expect (append (take lst 3) (list (nth-elem lst 3)) (drop lst 3)) lst)
```